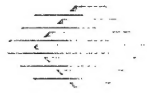


Carnegie Mellon
Software Engineering Institute

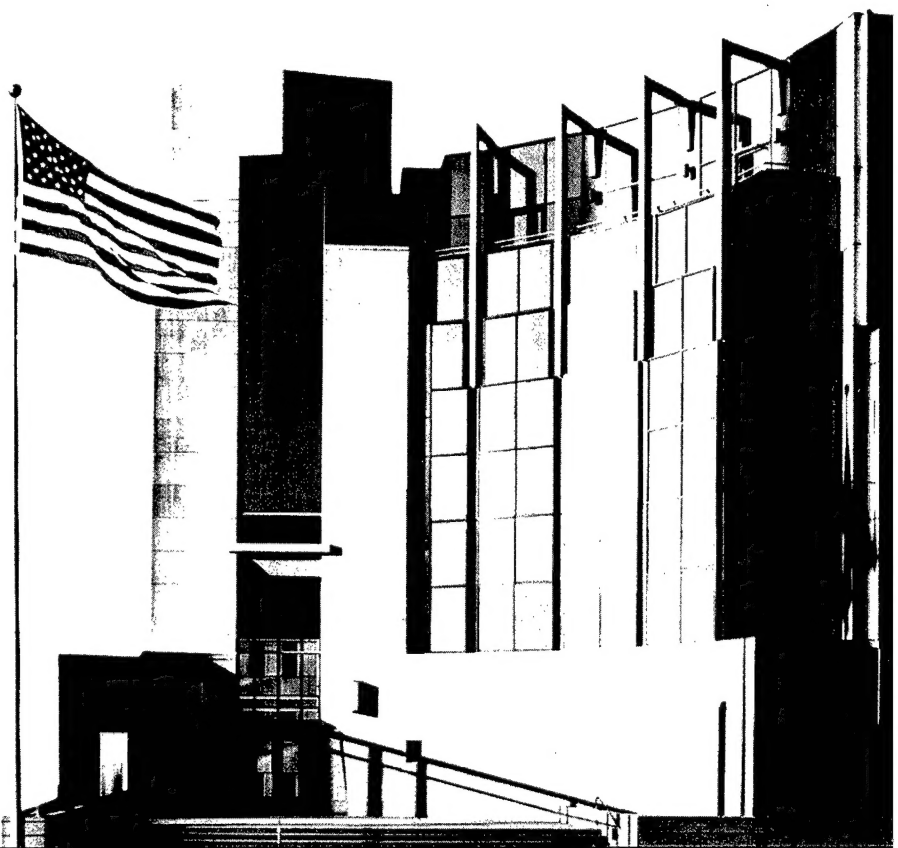
Replaceable Components and the Service Provider Interface

Robert Seacord
Lutz Wrage

July 2002

COTS-Based Systems

Technical Note
CMU/SEI-2002-TN-009



Carnegie Mellon University does not discriminate and Carnegie Mellon University is required not to discriminate in admission, employment, or administration of its programs or activities on the basis of race, color, national origin, sex or handicap in violation of Title VI of the Civil Rights Act of 1964, Title IX of the Educational Amendments of 1972 and Section 504 of the Rehabilitation Act of 1973 or other federal, state, or local laws or executive orders.

In addition, Carnegie Mellon University does not discriminate in admission, employment or administration of its programs on the basis of religion, creed, ancestry, belief, age, veteran status, sexual orientation or in violation of federal, state, or local laws or executive orders. However, in the judgment of the Carnegie Mellon Human Relations Commission, the Department of Defense policy of "Don't ask, don't tell, don't pursue" excludes openly gay, lesbian and bisexual students from receiving ROTC scholarships or serving in the military. Nevertheless, all ROTC classes at Carnegie Mellon University are available to all students.

Inquiries concerning application of these statements should be directed to the Provost, Carnegie Mellon University, 5000 Forbes Avenue, Pittsburgh, PA 15213, telephone (412) 268-6684 or the Vice President for Enrollment, Carnegie Mellon University, 5000 Forbes Avenue, Pittsburgh, PA 15213, telephone (412) 268-2056.

Obtain general information about Carnegie Mellon University by calling (412) 268-2000.

Replaceable Components and the Service Provider Interface

Robert Seacord
Lutz Wrage

July 2002

COTS-Based Systems

Technical Note
CMU/SEI-2002-TN-009

Contents

List of Figures

List of Tables

Abstract

Several popular component-based standards have emerged recently, including JavaBeans® and Enterprise JavaBeans® from Sun Microsystems and the Component Object Model from Microsoft. These component models are being adopted for use in software development, as they eliminate opportunities for architectural mismatch and are supported by standard services. A highly touted property of component models is that they support the development of replaceable components. Unfortunately, a robust, commercial marketplace of replaceable components has not been established for any of these component models.

On the other hand, the properties of the Service Provider Interface (SPI), used in many Java language packages, have resulted in the development of reusable components in several technology areas. Examples of successful SPI packages include Java Database Connectivity, Java Cryptography Extension, Java Naming and Directory Interface, and the Java Application Program Interface for XML Processing.

This technical note considers the motivation for using replaceable components and defines the requirements of replaceable component models. It evaluates the properties of standard component models and the SPI approach that affect their ability to support replaceable components.

® JavaBeans and Enterprise JavaBeans are registered trademarks of Sun Microsystems.

1 Introduction

A highly touted property of components and component-based software engineering is the ability to treat components as fully replaceable units. Philippe Krutchen of Rational Software, for example, has defined a software component as a nontrivial, nearly independent, and replaceable part of a system that fulfills a clear function in the context of a well-defined architecture [Kruchten 98].

Beyond eliminating some forms of architectural mismatch, commercially successful component models such as JavaBeans®, Enterprise JavaBeans (EJB)®, and Component Object Model (COM)[1] do little to support or encourage components as replaceable units. These component models do not impose any kind of functional interfaces on conforming components. Consequently, replaceable components that implement these component models have not materialized in the marketplace. Sun Microsystem's service provider interface (SPI), on the other hand, has been considerably more successful in this regard—encouraging the creation of replaceable components in a handful of technology areas, including cryptographic service providers, naming and directory services, and database connectivity. Additional SPIs are also being developed (or matured) for printing services, XML parsing and other technologies.

This paper examines the characteristics of Java's SPI, analyzes why SPIs have generated a market of reusable components while standard component models have not, and considers what value (if any) these other component models hold over the use of SPIs.

This document is organized into the following sections:

- Section 2: a definition of replaceable components with a focus on possible motivations for replacing a software component.
- Section 3: a very short overview of existing component models.
- Section 4: four examples of replaceable components and service provider interfaces that are specified by Sun
- Section 5: a high-level comparison of the four examples from section 4
- Section 6: conclusion

® JavaBeans and Enterprise JavaBeans (EJB) are registered trademarks of Sun Microsystems.

[1] For the purpose of this document, COM includes Microsoft's component technologies Distributed Component Object Model (DCOM) and COM+.

2 Replaceable Components

A replaceable component is one for which another component can be substituted without substantial modification to the new component or the existing system.[2] In considering a component model for replaceable components, it is necessary to first understand the motivations for supporting replaceable components. This understanding is necessary before a replaceable component model can be defined or recognized. The following questions promote this understanding:

- Is it necessary or useful to reinstall a component that has already been replaced? By this we mean a scenario analogous to lenses for photo cameras. A user has replaced component A with component B in a software system. Is it then necessary or useful to keep component A in store and replace component B again with A later on? In this case A and B are used as alternatives in specific situations.
- Should components be replaceable by the end user of the system? By the system administrator or by the system developer or maintainer?
- Must components be replaceable at runtime?

In the following sections we draw parallels between physical systems and software systems in an attempt to expose the motivations for supporting replaceable components.

2.1 Physical Systems

The idea of replaceable components has obvious physical system parallels. Consumer products such as automobiles, vacuum cleaners, stoves, and refrigerators all contain replaceable components. For the most part, these products are engineered with replaceable components because

- It is costly to replace the overall system.
- Some system components tend to wear and break before other components.
- It is economical to replace failing parts when compared to replacing the entire system.

Electronic products such as cameras, stereos, and computers use replaceable parts not only because these components can fail but also to modify or enhance the functionality of these systems. For example, a photographer may wish to replace a 50mm camera lens with a

[2] A degenerate case exists when the new component is a modified version of the original component. Typically, this case is considered an upgrade and not a replacement. To be considered fully replaceable, substitute components should be available from a vendor other than the developer of the original component.

telephoto or wide-angle lens. Similarly, a computer user may decide to upgrade to a higher resolution monitor, or a music enthusiast may decide to upgrade a speaker system.

Another reason many consumer products have replaceable components is for flexibility in manufacturing. Systems with replaceable components can be easily reconfigured for different price points by selecting components of varying functionality, quality, and cost. Manufacturers can also incorporate different components when a principal supplier is not able to meet a need.

In summary, the reasons for replacing components in physical systems include

- replacing failing components
- enhancing or modifying system functionality
- providing manufacturing flexibility

2.2 Software Systems

While it is interesting to explore physical system parallels, it is often dangerous to draw analogies between software and hardware components since each class of component has significantly different qualities. As a result it is necessary to consider each reason in turn to determine if the analogy holds.

The first reason, to replace failing components, immediately appears to break the physical systems analogy. Unlike fan belts in automobiles, software components are not subject to failure from wear and tear. However, fielded components may eventually become obsolete, as their environment changes around them. Platform, operating system, and middleware upgrades (as well as upgrades to other components of the system) may make it necessary to replace an existing component. The motivation for replacing the software component is often the same as the motivation for replacing a failing part—simply to maintain the existing system's functionality.

The principal drivers in replacing a "failing" software component are cost and transparency. In particular, replacing the component should not cause the redevelopment or addition of a significant amount of code in the existing system, or require the new component to be modified in any way. This is typically accomplished by obtaining a more recent version of the product from the vendor, either through a maintenance contract or a new purchase. In this case, support for replaceable components exceeds our requirements—it is sufficient for the vendor to provide an upgrade path. There are occasions, however, in which the component vendor is no longer developing new versions of the product. In these cases a replaceable component is a decisive benefit.

The second reason given for replacing a component is to enhance or modify the functionality of a system. In the case of a camera, different lenses may be attached to a camera for

different purposes. All share a standard interface with the camera, and can communicate information such as the *f*-stop and shutter speed. There are tradeoffs involved, however, in the selection of a lens. A telephoto lens may provide for greater magnification but require additional lighting. As a result, no single lens provides the optimal set of qualities for all situations.

From a user perspective, being able to replace the lens has the advantage of changing the functionality of the camera without significantly changing the size or weight. If size and weight were not an issue, for example, wouldn't it be better to have several lenses attached to the camera and simply rotate a wheel to use a different lens? This would allow all of the lenses to be available all of the time.

An example of a software component is a piece of software that provides an encrypted communications channel. Different encryption components may provide the same basic functionality but differ in quality of service (for example, encryption methods, key lengths, or throughput). So why would we want to replace these components rather than simply add new ones? One reason is again size, in memory, and on disk. If size were not an issue, a software solution such as "add-ins" would be more appropriate—allowing us to extend, rather than replace, the capabilities of an existing tool or product. There are also situations in which an organization or user simply has no need for multiple options, but needs to select an option that is compatible with existing systems or corporate policy. Replaceable components are well suited to each of these situations. Another even more important reason to replace this type of component is security. If a specific encryption algorithm can no longer be considered sufficiently secure, this algorithm should be removed from the software system to prevent users from inadvertently choosing this algorithm.

The third reason provided for supporting replaceable software components is flexibility in manufacturing. Software, of course, is not manufactured in the same sense as hardware systems, yet some similarities with software development remain. In particular, manufacturers may turn to an alternate supplier to reduce cost, incorporate a higher quality component, or because the original supplier could not supply the component in sufficient quantity. Unlike hardware components, software is extremely simple to replicate, and having a sufficient supply is never an issue. However, software component suppliers may discontinue a component or go out of business. In this case, the continuing supply of new *versions* of the software is threatened. Without this continuing supply of new component version, the long-term maintainability of the system becomes threatened, requiring the replacement of the original component.

The use of replaceable components to provide flexibility in manufacturing also has some interesting parallels in software product lines [Clements 01]. Support for replaceable components may lead to the development of applications that can be more quickly customized to a particular situation.

2.3 Component Model Properties

With the principal motivations for replacing components identified, it is possible to answer the questions posed earlier concerning the properties of a replaceable component model.

The first question is "Is it necessary or useful to reinstall a component that has already been replaced?" The answer depends at least in part by what we mean by "replaced." In maintaining a system that uses multiple components, the replacement of a component or a tool (such as a compiler) is always treated as a major risk to the reliability, functionality and performance and other qualities of the system. Normally, a configuration branch is created in which the new component is introduced and any necessary work to integrate the replacement component is performed. Once the integration is completed, significant effort is exerted to revalidate the operation of the overall system. Eventually, the configuration management board will examine the evidence to determine if the component replacement has been successful, and if so, promote the configuration branch containing the replacement component. In answering this first question, it is assumed that the component is not considered to be replaced until this promotion occurs.

We have not identified "usefulness of reinstallation" as a motivation for replacing components nor as a requirement for a replaceable component model. Obsolete components or components with diminished functionality are unlikely to be reinstalled. Product lines normally start with a base system that is then customized in a particular direction. Each customization represents a separate evolutionary branch, so while one component may be installed in one branch while a different component is installed in a different branch, it is unlikely that an already-replaced component would be reinstalled in the same product line branch.

The second question is "Who should replace components? The end-user of the system, the system administrator, or the system developer or maintainer?" and the third question is "Must components be replaceable at runtime?" None of our motivations for replacing components suggests that either the system administrator or end-user of the system would need to replace components. In all cases, these changes could be made by the system developer and pushed out to the end-users of the system. Similarly, there is no real requirement for components to be replaced at runtime. Of course, a component model that allowed end-users to replace components at runtime would be preferable to one that did not, but this should not be considered a necessary condition of a component model that supported replaceable components.

While there is no requirement for end users to replace components at runtime, there is a requirement that the replaceable and replacing components share a similar, if not identical set of interfaces. Unless this is true, modifications will be required to the existing system or to the new component to resolve the mismatch, preventing simple replacement. It is often permissible for the new component to add interfaces, as long as the legacy interfaces are fully supported. It is also important to realize that non-functional "interfaces" such as memory

usage and latency may require maintenance for the system to continue to function properly after the replacement component has been installed.

3 Component Models

Component models such as Enterprise JavaBeans, JavaBeans, and Component Object Model (COM) have not stimulated a marketplace of replaceable components. For the most part, this is because these component models are designed to be general and support the development of a broad range of components. In this section we provide some background on Enterprise JavaBeans, JavaBeans, and COM and describe what these component models do (or do not do) to support the development of replaceable components.

3.1 Enterprise JavaBeans

Enterprise JavaBeans (EJB) provides a component model for server-side components that are typically transactional and often need to be secure. As a result, the component model integrates transactions and security services into the framework, allowing these capabilities to be easily supported by the system.

Enterprise beans are required to provide certain interfaces, but these interfaces exist largely to support life-cycle management. The bean provider defines the functional Application Program Interface (API) exported by the enterprise bean. The functionality supported by an enterprise bean is not constrained by the EJB specification, and not restricted by the EJB server.

There exists the possibility that independently developed specifications define APIs for replaceable enterprise beans, but these fall outside of the EJB specification. Beyond eliminating the potential for architectural mismatch between enterprise beans (and between enterprise beans and EJB servers) the EJB component model does little to support the development of replaceable components.

3.2 JavaBeans

The JavaBeans component model is primarily used for developing graphical user interface (GUI) components and controls. The three most important features of a JavaBean are the set of properties it exposes, the set of methods it allows other components to call, and the set of events it fires.

Properties are named attributes associated with a bean that can be read or written by calling appropriate methods on the bean. The methods a JavaBean exports are simply Java class methods that can be called from other components or from a scripting environment. Events

provide a way for one component to notify other components that something interesting has happened. Under the event model an event listener object can be registered with an event source. When the event source detects that something interesting happens it calls an appropriate method on the event listener object.

The JavaBeans specification imposes no interface restrictions on individual JavaBeans, but simply defines a mechanism for integrating JavaBeans, and for accessing their properties, methods, and events from integration tools (such as a GUI builder). Because of this, JavaBeans does not impose sufficient interface constraints to support a replaceable component.

3.3 COM

COM is a binary compatibility specification and associated implementation that allows clients to invoke services provided by COM-compliant components (COM objects). Services implemented by COM objects are exposed through a set of interfaces that represent the only point of contact between clients and the object.

COM defines a binary structure for the interface between the client and the object. This binary structure provides the basis for interoperability between software components written in arbitrary languages. As long as a compiler can reduce language structures down to this binary representation, the implementation language for clients and COM objects does not matter—the point of contact is the runtime binary representation. Thus, COM objects and clients can be coded in any language that supports Microsoft's COM binary structure.

Similar to EJB and JavaBeans, COM is a general component model and makes no attempts to define specific functional interfaces that can be used to build replaceable components.

4 Service Provider Interfaces

The service provider interface is a mechanism used by Sun in the development of Java class libraries and standard extensions to the Java programming language. SPI has not been promoted by Sun as a component model, but does exhibit some interesting properties that lead to the development of replaceable components. SPIs have been used by Sun in the development of database connectivity, cryptographic service providers, and naming and directory services. The Java Application Program Interface for XML parsing also includes a "plugability" layer that, while not described as a SPI, provides a similar capability.

This section of this paper examines some SPIs in detail. The next chapter examines patterns in the various SPIs and contrasts SPIs with the component models already discussed.

Implementation Background. The implementation of SPIs is based on abstract classes and Java interfaces. The Java Naming SPI (`javax.naming.spi`), for example, contains a number of interfaces that must be implemented by the service provider. A Java interface consists of a set of constant definitions and method declarations without implementations (abstract methods). Interfaces are implemented by classes, and each class can implement multiple interfaces. Variables in the end-user application are declared using the interface data type. These variables can then reference any object implementing that interface, and any methods defined in the interface can be accessed. For example, a variable of type `Context` can be used to invoke any methods defined by the `Context` interface on any object, provided by any service provider, that implements the interface.

This use of interfaces allows the Java program to invoke methods on an object whose implementing class is not known at compile time. At runtime the Java class loading mechanism is used to dynamically locate and load classes that implement the SPI.

In addition, the service provider interfaces make heavy use of factory methods and factory classes to instantiate implementation classes in the service provider that are not known during compilation of a Java program.

4.1 Data Access with Java Database Connectivity

The Java Database Connnectivity (JDBC) Data Access API provides Java applications with access to persistent data sources. Originally designed to provide connectivity only to SQL databases, it has evolved to also include other forms of tabular data, such as spreadsheets and flat files. Java applications use the JDBC API to connect to data stores, store and retrieve data via SQL statements, and to access database metadata.

The JDBC architecture consists of an API that can be used by Java applications—to access a database, for example. A JDBC driver provides the implementation of the JDBC API. These JDBC drivers enable the actual communication with a database. Drivers are provided by most database vendors and as third-party products. The details of the driver's implementation and the communication mechanism are specific to the driver and the underlying database.

The introduction of JDBC as a common interface to SQL databases is an early attempt to define a common, standardized Java interface to a service that exists in very different implementations. Although the specification does not mention the term *service provider interface* explicitly, the set of interfaces that are implemented by a driver provides effectively the same capability as SPIs referenced in later specifications.

The JDBC specification distinguishes the following four general categories of drivers:

1. Type 1 drivers implement the JDBC API as a mapping to another data access API, such as Open Database Connectivity (ODBC). Drivers of this type are generally dependent on a native library, which limits their portability. The JDBC-ODBC Bridge driver is an example of a Type 1 driver.
2. Type 2 drivers are written partly in the Java programming language and partly in native code. These drivers use a native client library specific to the data source to which they connect. Again, because of the native code, their portability is limited.
3. Type 3 drivers use a pure Java[3] client and communicate with a middleware server using a database-independent protocol. The middleware server then communicates the client's requests to the data source.
4. Type 4 drivers are pure Java and implement the network protocol for a specific data source. The client connects directly to the data source.

Figure 1 shows a Type 4 driver on the left and a Type 3 driver on the right.

[3] "Pure Java" refers to programs that only rely on the documented and specified Java platform, that is, no native methods and no external dependencies aside from the Java Core APIs. Sun provides a certification process to ensure that code is in fact 100% Pure Java.

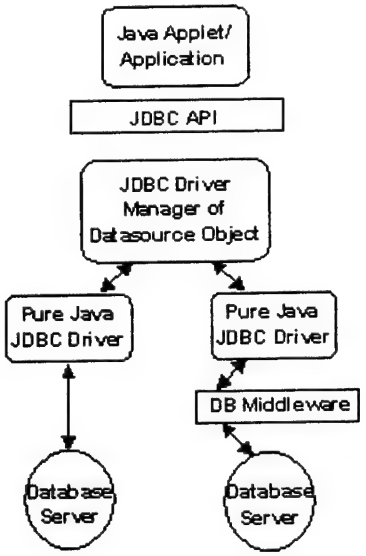


Figure 1: The JDBC Architecture Consisting of an API and a JDBC Driver

JDBC functionality is divided into a set of basic requirements, the `java.sql` package, and the `javax.sql` package. The basic requirements include a set of general features that a compliant JDBC implementation must support, such as Entry Level SQL92, transactions and access to all relevant database metadata. The `java.sql` package contains the set of core interfaces that represent specific features all JDBC driver implementations must support and additional, optional interfaces that should be supported if the underlying database supports the corresponding feature. The `javax.sql` package contains interfaces that represent optional, advanced features that may or may not be supported, such as

- connection and statement pooling
- Java Naming and Directory Interface (JNDI) support through data sources
- access row sets as JavaBeans
- distributed transactions

Replaceable Components. The packages `java.sql` and `javax.sql` of the JDBC API contain 12 class implementations and 31 interfaces that must be implemented by the driver developer. Most of these interfaces are partially or completely optional. This means that a driver implementation of a method in an optional interface may simply throw an exception. Java applications use a connection object to access a data store. Connectors can be obtained using the driver manager that works with one (or more) implementations of the `driver` interface or, preferably, by interacting with a data source implementation.

Working with the driver manager requires that the Java application first load the driver by instantiating and initializing a `Class` object for the JDBC driver's implementation of the driver interface. This implementation class is required to register an instance of itself with the driver manager in a static initializer that is called when the driver is loaded. The Java

application then uses the driver manager static "get connection" method to obtain a database connection. The code for doing this follows.

```
// Load the driver and initialize it
// oracle.jdbc.OracleDriver implements java.sql.Driver
Class.forName("oracle.jdbc.OracleDriver");

// Arguments for the database connection
String url = "jdbc:<vendor-specific string that identifies the
database>";

// Get a connection
Connection con = DriverManager.getConnection(
        url, "user", "password");
```

With this method, multiple JDBC drivers can be loaded simultaneously. In addition, as part of its initialization, the driver manager pre-loads all driver classes referenced in the `jdbc.drivers` system property. This allows a user to customize the JDBC Drivers used by their applications.

```
jdbc.drivers=com.db.DbJdbcDriver:com.xyz.XyzDriver
```

To allow the driver manager to select the appropriate driver for a database URL, the driver interface defines an "accepts URL" method that expects a URL as input parameter and returns a Boolean value. In this method the driver implementation checks whether it can handle the passed URL and indicates this to driver manager by returning true or false respectively. The "get connection" method queries registered drivers and uses one that can handle the passed URL.

The JDBC driver is replaced by replacing the class name of the driver implementation class. To reliably replace a driver at runtime, the previous driver can be de-registered with the driver manager. Replacing JDBC drivers is limited by the fact that most interfaces are optional. This makes it necessary to conduct a detailed comparison of supported features before using a different driver.

The second option for a Java application to obtain a connection to a data store is to retrieve a data source object that has been registered with a naming or directory service. This object represents a physical data source (for example, a database) and allows the Java application to get a connection object for this data source.

In this scenario the Java application developer has no influence at all on which JDBC driver is used for the database connection. This choice is made outside the application when the data source together with the appropriate JDBC driver is registered with the directory service. Only the name under which it is registered in the directory service must be known to the application, as the following illustrates.

```
// Get the initial naming context
Context ctx = new InitialContext();

// Arguments for the database connection
String dsName = "jdbc/DB";

// Get the data source and connection
DataSource ds = (DataSource) ctx.lookup(dsName);
Connection con = ds.getConnection("user", "password");
```

This method is used in Java 2 Enterprise Edition (J2EE) application servers to specify data sources for Enterprise JavaBeans and Web applications. In this case, it is particularly important that either the Java application makes no assumptions about the features of the data source, or that the choice of databases and JDBC drivers is restricted to those that provide a specific set of capabilities (for example, support for distributed transactions).

Locating JDBC Drivers. JDBC driver implementations are usually packaged as Java archives. A Java application loads these Java ARchives (JAR) files using either the default or a customized Java class loading mechanism. The default Java class loader is supplied with a list of directories and jar files (the class path) that indicates where and in which order to search for class files on a file system.

Service Provider Interface. The service provider interface of the JDBC API is the set of all interfaces that a JDBC driver can implement. It is identical to the set of interfaces that are available for use in a Java application. The database metadata interface contains variables and methods that permit the application to determine exactly which SQL features are supported.

4.2 Java Cryptography Extension

The Java Cryptography Extension (JCE) contains classes and interfaces for cryptographic services, such as encryption, key generation, key agreement, and Message Authentication Code (MAC) generation. While the JCE[4] was previously an extension to the Java platform, it has been integrated with Sun's Java Development Kit (Java 2 SDK) version 1.4.

Each cryptographic service is represented in the JCE by a Java class that provides access to the service, but does not include a concrete implementation of this service. These classes are called "engine classes" in the JCE specification. The cipher class, for example, provides access to the functionality of arbitrary encryption algorithms. Java applications use only the engine classes to access a cryptographic service. The following code snippet shows how a Java application may use DES encryption:

[4] Detailed information about the various releases of the Java Cryptography Extension is available at http://java.sun.com/products/jce/

```
// Generate a default length key using key generator
KeyGenerator g = KeyGenerator.getInstance("DES");
SecretKey k = g.generateKey();

// Get DES implementation and initialize for encryption
Cipher c = Cipher.getInstance("DES");
cipher.init(Cipher.ENCRYPT_MODE, key);

// Will transparently encrypt data written to stream
CipherOutputStream s = new CipherOutputStream(
        someOutputStream, cipher);
```

A cryptographic service provider that is plugged into the JCE implements underlying algorithms; for example, those used for encryption and key exchange.

Replaceable Components. The JCE framework does not specify which algorithms a cryptographic service provider must implement. Instead, it prescribes a mechanism that allows providers to announce names and properties of algorithms they implement. Typically, multiple providers that may implement different algorithms will be installed at the same time in a given preference order. The Java application can then request a specific algorithm or specific encryption strength, and the JCE chooses the appropriate implementation, if one is available.

To accomplish this, each provider must be implemented as a subclass of Java security provider, the provider's "master class," which encapsulates a set of properties (name-value pairs). The constructor of this subclass registers the provider's name and sets the values of various properties that indicate which algorithms the provider implements and the class name of the algorithm. A provider for the previous example contains the following code.

```
// Set encryption algorithm implementations
put("Cipher.DES", "com.acme.DESCipher");

// Set key generator implementations
put("KeyGenerator.DES", "com.acme.DESKeyGenerator");
```

When a Java application wants to use an algorithm, it calls the "get instance" method to get the name of the engine class (for example, "cipher") with the name of an algorithm as a parameter (for example, "DES"). This method constructs the property name and then queries installed providers in preference order for a matching property name. If such a property is found, the method creates an object of the corresponding class and returns it to the application for further use.

Alternatively, an application can request an implementation from a specific provider by passing the provider name as an additional parameter. A list of all installed providers is available through the Java security class. This class also has methods to add and remove providers, and to rearrange the preference order of providers.

In the case of cryptographic services, it is preferable that providers *not* be replaceable at runtime without user control. It is essential that the user approves the providers available to an application.

Security Requirements. As cryptographic services are used to protect sensitive data, evidence must be provided that users can trust a service provider implementation. In addition, mechanisms must be present to prevent Java applications and especially Java applets from tampering with the installed providers. This should be possible without sacrificing the flexibility of a plug-in mechanism. This leads to some interesting consequences for the JCE service provider interface:

- A JCE provider must be signed using a code-signing certificate issued by a "trusted" JCE Code Signing Certification Authority (currently Sun and IBM only). When a provider's implementation of a JCE service is instantiated, the JCE framework checks that the JAR archive containing the implementation is signed and verifies the signature. This prevents arbitrary, unapproved providers from being plugged into the JCE.
- A provider should follow several recommendations that prevent tampering with its implementation. The recommended measures include self-integrity checking and using Java language features to prevent applications to instantiate provider classes directly.
- Operations for manipulating the list of installed providers are privileged and as such subject to access control according to a security policy.
- The JCE consists of abstract classes with protected methods instead of interfaces. This prevents applications from circumventing an engine class and directly accessing the provider's algorithm implementation.

In addition, the JCE framework includes a means to reliably limit the cryptographic strength of algorithms available to applications to comply with U.S. export restrictions and with import restrictions of other countries.

Installing Cryptographic Service Providers. The first step in installing cryptographic service providers is to place the provider JAR file in the file system and to modify the class path, if necessary. In addition, the provider's master class must be added to the list of approved providers in the `java.security` text file that is part of the Java runtime environment. For each provider, this file has a statement that specifies the preference order *n* and the name of the provider's master class.

```
security.provider.n=com.acme.security.MasterClass
```

The JCE framework automatically registers all providers that have an entry in this configuration file. If an application must register a provider dynamically using the above-mentioned `Security` class, this can only be done by code that is granted the following permission.

```
java.security.SecurityPermission
        "insertProvider.<provider name>"
```

Additional permissions must be granted whenever a cryptographic service provider is used in the presence of a security manager, which is typically the case for applets or applications that use remote method invocation.

If two or more installed providers implement the same algorithm, the JCE framework will choose the implementation of the provider with the highest preference order.

Service Provider Interface. The service provider interface for cryptographic service providers includes a set of abstract classes in the packages `java.security` and `javax.crypto` and sub-packages of these. For each engine class *Engine* in these packages, there is a corresponding abstract class named *Engine*`Spi` that must be subclassed by a cryptographic service provider.

4.3 Java Naming and Directory Interface

The Java Naming and Directory Interface (JNDI) is a standard extension to the Java platform, providing Java applications with a single interface to heterogeneous enterprise naming and directory services.

The JNDI architecture consists of an application programming interface and a service provider interface. Java applications use the JNDI API to access a variety of naming and directory services. The SPI enables a variety of naming and directory services to be "plugged in," allowing the Java application using the JNDI API to access their services as shown in Figure 2.

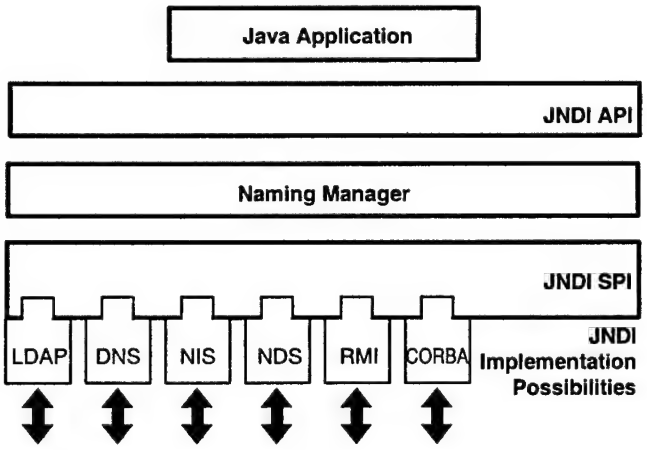


Figure 2: The JDNI Architecture Consisting of Both an API and an SPI

To use the JNDI, you must have the JNDI classes and one or more service providers that map the JNDI API to actual calls to the naming or directory server. The Java 2 SDK, version 1.4, includes four service providers for the following naming/directory services:

- Lightweight Directory Access Protocol (LDAP)

- Common Object Request Broker Architecture (CORBA) Common Object Services (COS) name service
- Java Remote Method Invocation (RMI) Registry
- Internet Domain Naming System (DNS)

Service providers exist for other naming and directory services as well, including the remote method invocation registry, network information service (NIS), directory services markup language (DSML), Novell Directory Service (NDS), file system, and the Windows Registry. But how does the SPI support replaceable components?

Replaceable Components. Before performing any operation on a naming or directory service, an initial context must be acquired to serve as the starting point into a namespace. The LDAP service must be able to determine which service provider to use to create the initial context. This is accomplished by creating a set of environment properties to which the name of the service provider class is added. For example, if you are using the LDAP service provider from Sun Microsystems, then your code appears as follows.

```
Properties env = new Properties();
env.put(Context.INITIAL_CONTEXT_FACTORY,
        "com.sun.jndi.ldap.LdapCtxFactory");
```

Once set, the environment is passed as an argument to the `InitialContext()` constructor to create the initial context:

```
Context ctx = new InitialContext(env);
```

By simply specifying a different class as the initial context factory, a different service provider (component) is installed—supporting replaceable components that can be replaced dynamically at runtime.

Federation. In addition to supporting replaceable components, JNDI also has a capability that more closely resembles the add-ins discussed earlier in this paper. Multiple service providers can be used by a single application in conjunction to support *composite namespaces*. Composite namespaces incorporate multiple naming systems, as shown in Figure 3. In this case, DNS is used as the global naming system. This name space is then split divided between NDS and LDAP.

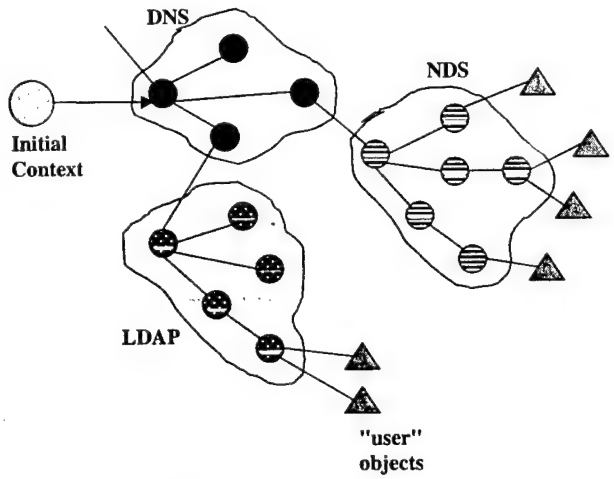


Figure 3: A Composite Namespace

A *composite name* may span multiple namespaces, that is, it consists of one or more *compound names*, each belonging to a single namespace. The components of a composite name are separated by a forward slash ("/") character. The context implementation must determine which part of the name is to be resolved in its context and pass the rest onto the next context. This may be done syntactically by examining the name, or dynamically by resolving the name. The resolution of a (multi-component) composite name proceeds from one naming system to the next, with the resolution of the components that span each naming system typically handled by a corresponding context implementation. From a context implementation's point of view, it passes the components for which it is not responsible to the context implementation of the next naming system.

There are several ways in which the context implementation for the next naming system may be located. It may be located explicitly through the use of a junction, where a name in one naming system is bound to a context (or a reference to a context) in the next naming system. For example, with the composite name `cn=fs,ou=eng/lib/xyz.zip`, the LDAP name `cn=fs,ou=eng` might resolve to a file system context in which the name `lib/xyz.zip` could then be resolved.

Alternately, the next naming system may be located implicitly. This can be done statically in the context implementation, or dynamically by the framework. For example, with the composite name `ldap.sei.cmu.edu/cn=fs,ou=eng`, the DNS name `ldap.sei.cmu.edu` might name a DNS entry. If the DNS Context implementation cannot determine the next naming context, it constructs an exception that contains the result of the resolution so far, and the unresolved rest of the composite name. The `NamingManager` class in the JNDI framework contains a method `getContinuationContext` that finds an appropriate "next" context in which the resolution can continue. A corresponding method is provided for accessing a directory service.

How ever the next naming system is located, the remaining portion of the composite name is handed to the context implementation to resolve.

Locating Service Providers. Let us review what we have learned so far. Each service provider implements a context factory object that supports JNDI operations with a name space. An initial context can be established using the `InitialContext()` constructor. Name spaces can be federated, using composite names and a variety of mechanisms for locating the next context. The only outstanding question is "How does JNDI know how to locate service providers that could potentially provide the context within which a name can be resolved?"

The answer is relatively simple. Each deployable component is responsible for listing the factories that it exports. Each service provider has an optional resource file that contains properties specific to that provider. The name of this resource is

```
[prefix/]jndiprovider.properties
```

When an application is deployed, it will generally have several code base directories and JARs in its class path. Similarly, when an applet is deployed, it will have a code base and archives specifying where to find the applet's classes. JNDI locates all application resource files named `jndi.properties` in the class path. These files are searched for lists of JNDI factories, which are concatenated into a single colon-separated list. For example, if the `java.naming.factory.object` property is found in three `jndi.properties` resource files, the list of object factories is a concatenation of the property values from all three files.

Service Provider Interface. The service provider interface to be implemented by a JNDI provider consists of interfaces only. The interfaces that support plug-in of JNDI providers into the JNDI framework (mostly factory classes) are grouped in the `javax.naming.spi` package. Additional interfaces that must also be implemented provide Java applications with access to the providers' services. These are defined in the `javax.naming` package.

4.4 Java API for XML Processing

The Java API for XML[5] Processing (JAXP) provides basic support for parsing and manipulating XML documents through a standardized set of Java Platform APIs. JAXP does not explicitly claim to provide a SPI, but nonetheless, the interface provides a similar (if not identical) capability.

[5] The eXtensible Markup Language (XML) is the metalanguage defined by the World Wide Web Consortium (W3C) that can be used to describe a broad range of hierarchical markup languages. It is a set of rules, guidelines, and conventions for describing structured data in a plain-text, editable file.

JAXP defines plugability interfaces for SAX[6], DOM[7], and XSLT[8]. The plugability interfaces to SAX and DOM allow access to the functionality defined in the SAX 2.0 API[9] and DOM Level 2 specification[10] respectively, while allowing the choice of the implementation of the parser. Depending on the needs of the application, JAXP provides developers the flexibility to swap between XML processors (such as high-performance vs. memory conservative parsers) without making application code changes. The plugability interface for XLST allows an application programmer to obtain a transformer object that is based on a specific XSLT style sheet.

All three JAXP plugability interfaces are implemented in a similar fashion, so any one of them is representative of the plugability interface. The SAX Plugability interface is as good an example as any to consider.

SAX Plugability. The SAX plugability classes allow an application programmer to provide an implementation of the default handler API to a SAX parser implementation to parse XML documents. As the parser processes the XML document, it will call methods on the provided default handler.

To obtain a SAX parser instance, an application programmer first obtains an instance of a SAX parser factory. The SAX parser factory instance is obtained via the static new instance method of the SAX parser factory class.

The new instance method uses the following ordered look-up procedure to determine the SAX Parser Factory implementation class to load:

1. Use the `javax.xml.parsers.SAXParserFactory` system property.
2. Use the properties file lib/jaxp.properties in the JRE directory. This configuration file is in standard Java properties format and contains the fully qualified name of the implementation class, with the key being the system property defined above.
3. Use the Services API (as detailed in the JAR specification), if available, to determine the class name. The Services API will look for the class name in the file `META-INF/services/javax.xml.parsers.SAXParserFactory` in jars available to the runtime.
4. Use the Platform default SAX Parser Factory instance.

---

[6] The Simple API for XML (SAX) is a public domain API that provides an event-driven interface to the process of parsing an XML document. An event-driven interface provides a mechanism for "callback" notifications to application's code, as the underlying parser recognizes XML syntactic constructions in the document.

[7] The Document Object Model (DOM) is a set of interfaces defined by the W3C DOM Working Group. It describes facilities for a programmatic representation of a parsed XML (or HTML) document.

[8] The XSL Transformations (XSLT) describes a language for transforming XML documents into other XML documents or other text output defined by the W3C XSL Working group.

[9] The SAX 2.0 API is located at *http://www.saxproject.org*.

[10] The DOM Level 2 Core Recommendation is located at *http://www.w3.org/TR/2000/REC-DOM-Level-2-Core-20001113/*.

5 SPI Comparison

There are similarities and differences in capabilities that can be noted between the SPIs examined in this report.[11] For example, the JDBC API provides a uniform interface to various data sources, including SQL databases. A JDBC driver serves as a connector between a Java application and some data storage technology. Data storage technologies vary widely with regard to supported features and support of standards. Many database vendors include features that are unique to their products or provide extensions to standards. In addition, JDBC driver implementations for the same database but by different vendors do not necessarily provide the same level of access to the capabilities of this database. Since database access is a critical factor for an application's performance, the JDBC API cannot restrict drivers to some least common denominator.

Similarly, the JNDI connects Java applications to external services, but these services have interfaces and features that are well defined compared to database systems. This is especially true for implementations of the LDAP standard. In addition, the number of different naming and directory services is limited, and the feature set does not vary as widely as it does for databases. An important feature of the JNDI API is the support for federation that provides Java applications with a unified view of an environment that integrates different directory and naming services.

JCE follows a different model than both the JDBC and JNDI. JCE is not concerned with connecting technologies but with providing a plug-in mechanism for algorithms. These algorithms provide implementations of pre-defined classes of cryptographic services. Different providers may complement each other by supporting different algorithms or the same algorithms with varying cryptographic strengths.

The JAXP API describes a similar plug-in mechanism, in this case for XML parsing and XSL transformation services provided by a compliant parser or transformer. The JAXP API was originally developed to repair deficiencies in the existing SAX and DOM industry standards. These repairs include bootstrapping a DOM tree from an XML document and controlling parser validation. The transformation part of the API has been added in the latest version for completeness.

All four described SPIs meet the minimal conditions for supporting a replaceable component model outlined earlier in this paper. In addition, they all support runtime replacement of

[11] These differences are certain to exist and should not be considered a criticism, given that the SPI is not a standardized component model, but more of a generic mechanism that has been used on multiple occasions by Sun developers.

components by the end user of the system, through modification of system properties to install a different service provider. The JDBC, JNDI, and JCE SPIs go beyond replaceable components to support an add-in capability. In other words, multiple service providers can be installed simultaneously with the correct service provider being invoked based on a database URL, a composite name, or other factors described in this paper. JAXP has no need to provide this capability (and does not) since a single XML parser is sufficient in almost any case.

Table 1 summarizes the properties of the four evaluated examples:

	Provider replaceable at runtime	Add-in of providers	Explicit SPI	Connector mechanism	Plug-in mechanism
JDBC	✓	✓		✓	
JNDI	✓	✓	✓	✓	
JCE	✓	✓	✓		✓
JAXP	✓				✓

Table 1: Properties of Different SPIs

5.1 Market

When considering the ability of a component model (or SPI) to generate a market for replaceable components, it would be negligent not to look at the market each SPI has generated at the time this report was written.

For JDBC there is a survey on Sun's Web site that lists available JDBC drivers together with the supported specification version and the implemented optional features. The list contains 50+ vendors that offer 75+ products. The list of vendors includes almost all major database vendors that offer drivers for their database products (usually for free). In addition, it also contains third-party vendors that offer JDBC drivers for multiple databases, often with integrated value-added features, such as encrypted connections or advanced monitoring capabilities, for example.

The other SPIs have generated much smaller markets. There are about 10 JCE implementations by about 10 vendors, including several open-source projects. For JNDI, there are 10 products from 5 vendors, and for JAXP there are just 5 products from 3 vendors, including the Apache Group that supplies 2 parsers and one transformer (crimson, xerces, and xalan).

A number of reasons might explain why these SPIs have failed to create significant markets; some of these result from the problem these SPIs address. This is most obviously the case for JNDI because a JNDI provider essentially forms a bridge between two standards. Once an implementation exists, there is no good reason to provide an additional one, as the existing

solution can be reused elsewhere without problems. The LDAP providers, for instance, that are often shipped in connection with J2EE application servers, can be the same that are included in Sun's Java runtime environment (for example, BEA Weblogic). Moreover, there is not much opportunity for adding bells and whistles to a JNDI provider that would justify the effort to market it as a separate product.

While this is certainly not the case for cryptographic service providers, encryption is a complex subject, encumbered by country-specific export and import regulations. Additionally, the requirement to have a cryptographic service provider implementation signed via Sun or IBM sets an additional obstacle that might prevent software companies from implementing JCE providers.

The number of JAXP implementations is similarly driven by market conditions. Freely available products dominate the market for Java implementations of XML parsers and XSLT processors. Since XML and XSLT are standards that do not permit vendor specific extensions, the only potential advantage commercial offerings could have over the existing offerings (apart from support) is performance. But if performance really is an issue, parsers and processors not implemented in Java but in a compiled language such as C, for example, may be the preferred option anyway.

In summary, the number of replaceable SPI components in each market area is commensurate with the prevailing market conditions and not an indicator of the lack of suitability of the SPI model.

6 Component Model Comparison

Component models such as Enterprise JavaBeans, JavaBeans and COM do little to support the development of replaceable components. These component models impose interfaces on components that allow them to be manipulated in a standard fashion by component frameworks. However, these component models do not impose any kind of functional interfaces on components that conform to the model, making it impossible to replace these components without modification to the remaining system.

In many ways the service provider interface is a more effective mechanism for supporting replaceable components than any of the standard component models. This is because each SPI is focused on a particular functional area, and defines a common API that can support that functional area while still allowing for variations in *how* the functionality is implemented. Standard component models are meant to be more general and do not provide this level of functional interface specification. This is not a criticism of these component models. In fact, the major point of this paper may be that expecting these general component models to generate a marketplace of replaceable components is unreasonable.

References

[Clements 01] Clements, Paul & Northrop, Linda. *Software Product Lines*. Boston, Ma.: Addison-Wesley, July 2001.

[Comella-Dorda 97] Comella-Dorda, Santiago. *Component Object Model (COM), DCOM, and Related Capabilities.* <http://www.sei.cmu.edu/str/descriptions/com_body.html > (1997)

[Kruchten 98] Kruchten, Philippe. "Modeling Component Systems with the Unified Modeling Language," in *Proceedings of the 1998 International Workshop on Component-Based Software Engineering,* Kyoto, Japan, 1998. <http://www.sei.cmu.edu/cbs/icse98> (1998).

[Long 98] Long, F. & Seacord, R.C. "A Comparison of Component Integration Between JavaBeans and PCTE," in *Proceedings of the 1998 International Workshop on Component-Based Software Engineering* Kyoto, Japan, 1998. <http://www.sei.cmu.edu/cbs/icse98> (1998).

[Rajiv 01] Rajiv, Mordani; Davidson, James Duncan; & Boag, Scott. *Java API for XML Processing Version 1.1* Final Release, Sun MicroSystems, February 2001. http://java.sun.com/xml/jaxp/dist/1.1/ jaxp-1_1-spec.pdf> (2001).

[Sun 95] Sun Microsystems. JDBC Data Access API. <http://java.sun.com/products/jdbc/overview.html> (1995-2002).

[Szyperski 98] Szyperski, C. & Vernik, R. "Establishing System-Wide Properties of Component-Based Systems," *Proceedings of OMG-DARPA-MCC Workshop on Compositional Software Architecture.* Monterey, Ca., January 1998.

[Szyperski 98] Szyperski, C. *Component Software Beyond Object-Oriented Programming*. Boston, Ma.: Addison-Wesley and ACM Press, 1998.

[Wallnau 01] Wallnau, K.; Hissam, S.; & Seacord, R. *Building Systems from Commercial Components*. Boston, Ma.: Addison-Wesley, July 2001.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
(Leave Blank)	July 2002	Final

4. TITLE AND SUBTITLE
Replaceable Components and the Service Provider Interface

5. FUNDING NUMBERS
F19628-00-C-0003

6. AUTHOR(S)
Robert Seacord, Lutz Wrage

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Software Engineering Institute
Carnegie Mellon University
Pittsburgh, PA 15213

8. PERFORMING ORGANIZATION REPORT NUMBER
CMU/SEI-2002-TN-009

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
HQ ESC/XPK
5 Eglin Street
Hanscom AFB, MA 01731-2116

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

12A DISTRIBUTION/AVAILABILITY STATEMENT
Unclassified/Unlimited, DTIC, NTIS

12B DISTRIBUTION CODE

13. ABSTRACT (MAXIMUM 200 WORDS)

Several popular component-based standards have emerged recently, including JavaBeans® and Enterprise JavaBeans® from Sun Microsystems and the Component Object Model from Microsoft. These component models are being adopted for use in software development, as they eliminate opportunities for architectural mismatch and are supported by standard services. A highly touted property of component models is that they support the development of replaceable components. Unfortunately, a robust, commercial marketplace of replaceable components has not been established for any of these component models.

On the other hand, the properties of the Service Provider Interface (SPI), used in many Java language packages, have resulted in the development of reusable components in several technology areas. Examples of successful SPI packages include Java Database Connectivity, Java Cryptography Extension, Java Naming and Directory Interface, and the Java Application Program Interface for XML Processing.

This technical note considers the motivation for using replaceable components and defines the requirements of replaceable component models. It evaluates the properties of standard component models and the SPI approach that affect their ability to support replaceable components.

14. SUBJECT TERMS	15. NUMBER OF PAGES
CBSE, Component-Based Software Engineering, Enterprise JavaBeans, JavaBeans, EJB, COM, Component Object Model, SPI, Service Provider Interface	35

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	UL